A guide for the assembly-language programmer on the move

OS-9 Assembly Language

by Jeff Mikel

After several years of programming with Radio Shack's EDTASM+ assembler. I finally got OS-9 Level II and the OS-9 Level II Development System. This gave me access to all those things I dreamed of doing — editing files larger than 32K. editing and assembling large programs in memory, and easy disk, printer and keyboard I/O.

But moving from EDTASM+ to OS-9 was not automatic. I had a great deal to learn about OS-9 assembly-language programming. The OS-9 environment is an assembly-language programmer's dream. But with all the extra capability comes a set of design rules that must be followed if your programs are to perform properly. This is a small price to pay to reap the benefits of conditional assembly, macros, very large source files, and even local and global variables. If there is a high-level assembly language, the OS-9 Level II Development System has it.

Learning the new software packages was challenging since a lot of the necessary information is hidden in the manuals. It took a lot of manual searching and program writing before I found the information I needed.

This article covers the basics you need to know, but can't seem to find. First, I cover the assembler and linker of the Development System. Then, I discuss how to write new OS-9 commands. Finally, we'll take a brief look at the techniques for writing subroutines for BASIC09.

Throughout this article, I assume you are familiar with the CoCo's 6809 instruction set and that you have written enough assembly-language code to be comfortable with it. I do not attempt to teach the hows and whys of assembly-language programming. Rather, I explain how to write source code for the Relocating Macro Assembler or RMA. I also assume you understand enough about OS-9 to know how and when to change directories, specify pathlists and perform the other necessary functions without being told.

If you do not have much OS-9 experience, you should first get some background with BASIC09, or some other high-level language, to familiarize yourself with OS-9 operation. However, if you have used EDTASM+, or other packages under Disk BASIC, and are familiar with OS-9, you are primed and ready to forge ahead to OS-9 and the RMA package. And, if you are already familiar with the asm assembler, you can probably skim this article and study only the parts that relate exclusively to the RMA.

Building a Working Disk

Before assembling any programs, first set up a working disk with the necessary commands and enough space to hold your source files. Do not use the original Development System disks — put write protect tabs on them and keep them in a safe place.

Your working disk should have, at the very least, the files rma and rlink in the CMDS directory. If you plan to use scred an excellent text editor for assembly-language programming, make sure it is also in the CMDS directory, scred needs the file termset in /dd/SYS, so make sure that file is also available.

I recommend the sys.l file be on your working disk. This file can be used by the linker for resolving the system-wide symbolic names and is faster than the Level I assembler's solution of including the statement use os9defs in each of your source files, sys.l is found in the LIB directory on your master disk. I keep my copy of sys.l in the root directory to save keystrokes.

If you have 512K, you definitely want to use the RAM disk included with the Development System during the editing and assembly process. There are three sizes from which to choose. The 96K version is adequate for my assembly-language needs. The necessary modules for using the RAM disk are in the MODULES directory on your Development System disk.

A quick way to start using the RAM disk is to load it from your Development System disk each time you boot your system. The RAM disk comes in two parts: A device driver and a device descriptor. Both must be in memory for the RAM disk to work. To save memory, you should first merge the driver module with the device descriptor you want. For example,

to create a 128K RAM disk, load attr if it is not already in memory. Then, put your (backup) Development System disk into /d0 and enter

```
chd /d0/modules
merge ram.dr r0_128k.dd >/d0/ram disk
attr /d0/ramdisk e pe
```

From now on. you can insert the working disk into drive /d0 and enter

```
load /d0/ramdisk
iniz /r0
```

Now device /r0 functions like one of your disk drives, except it is much faster. Although this method is easy, it removes 8K from the OS-9 system memory, which could severely limit the number of active processes or devices at one time.

The best way to add the RAM disk to your system is to create a new boot disk with the device driver and device descriptor included in the os9boot file. You can also configure the RAM disk as device /dd instead of /r0 in this way. I like this method very much. My system boots with /dd as a 96K RAM disk. After booting I use a procedure file to set up my RAM disk with the required files. The following is a sample procedure file for a single-drive system:

```
load makdir
chx /d0/cmds
load scred
load rma
load rlink
-x
makdir /dd/CMDS
makdir /dd/SYS
unlink makdir
copy /d0/sys/termset /dd/sys/termset
copy /d0/lib/sys.l /dd/sys.l
```

With all the required files either in memory or on the RAM disk, the assembler and linker really fly.

Now that you have a working disk, it's time to explore the realm of OS-9 assembly language.

A Quick Review

Take a quick look at Listing 1. a short program that prints "Have a nice day." on the screen. Those of you who have attempted to write similar programs with EDTASM+ will appreciate the brevity of this program. I will quickly cover its operation.

The first line, which starts with psect, gives the assembler some information about the program. (I'll discuss that in more detail later.) The program begins at Start, where Register X is set to point to the output string. Next. Register Y is loaded with the length of the string in bytes. Register A is then loaded with 1. which is the path number for the standard output device (usually the screen). The os9 I$WritLn system call writes the string to the screen and F$Exit returns control to the Shell. Since Register B was cleared earlier, the Shell assumes no errors occurred and — voila! — a successful program.

The Assembler Directives

The assembler has a set of statements called program section directives that direct it (hence the name) in its production of executable code. These directives are: psect, csect and vsect. Of the three, the only required directive in any program is psect. Every program must have a psect.

The psect, or program section directive, marks the beginning of a code section. The assembler and the linker use the information given in the psect directive in building the final OS-9 module. Proper syntax for psect is

```
psect name, typelang, attrev,
edition, stacksize, entry
```

For Level I asm users, the psect directive takes the place of the mod directive. Each source file should have only one psect directive, but multiple psects (from multiple files) can be connected with the linker. This enables you to build a library of common routines, debug them once and use them forever.

The following is a description of each psect parameter:

name — the name for the psect. The psect name is arbitrary, but it is good practice to use a name that describes the purpose or function of the routines in the psect. Names can be as long as 20 characters. The assembler does not use this name, but the linker does when it reports errors. It is advisable to give each psect a unique name.

typelang — this specified value becomes the type/language byte for the final memory module. The four high-order bits define the module type and the four low-order bits specify the language type. Type and language codes are specified on Page 3-4 of the Technical Reference section in your OS-9 Level II manual.

attrev — this value becomes the attributes/revision byte for the final module. The four high-order bits specify the module's attributes. Currently, only Bit 7 is defined. When set. Bit 7 indicates the module is reentrant and can therefore be shared. The four low-order bits specify the module's revision number. edition — this becomes the edition byte in the module header.

stacksize — this is an estimation as to how much memory the routines in this psect need for the stack. Be generous; remember that OS-9 uses the stack even if your program does not.

entry — the entry address of your program: the address to which OS-9 transfers control when your program is executed.

With reference to Listing 1, the program section name is first, the module's type/language byte is $11 (a 6809 object-code program module), and the attributes/revision byte is $81 (a reentrant program with Revision 1). The program is a first edition with 100 bytes reserved for stack space. The program begins at Start.

Direct your attention to the sample program header in Listing 2. This listing demonstrates the use of symbolic names in the psect directive. The only differences between this psect and the one declared in Listing 1 are the name and stack size requirements. However, the real purpose of Listing 2 is to demonstrate the other two program section directives: vsect and csect.

The vsect, or variable section directive, tells the assembler to begin a variable storage section. There are two types of vsects: Direct-page and non direct-page. If you specify a direct-page, the linker assigns the variables that are defined within the vsect to the 6809's direct page (the page referenced in conjunction with the 6809's DP register). The first vsect in Listing 2 is a direct-page vsect, since vsect dp is used. Thus, the count and path variables are located in the direct page.

Variables stored in the direct page are more quickly accessed and require less program code than those not stored in the direct page. So, store your often-used variables there. Keep in mind there are only 256 bytes in the direct page, so you may not be able to employ the direct page for all the variables you are using.

The IObuffer and filename variables are not located in the direct page. These variables should not be accessed through the same procedure to which you are accustomed with EDT-ASM+ since neither you nor the assembler know the exact location of these variables at run time. Basically, don't use extended addressing. Use an index register and reference these variables as offsets from the base address.

Notice that the variables in the second vsect are large — large enough to use all the direct-page space had they been put there.

It is advantageous to keep these large variables off the direct page since it is prime real estate. You are not limited to just two vsects per psect — you can have as many vsects as necessary. It is advisable, but not absolutely required, that you put all similar variables in a single vsect at the beginning of the program.

csect is the next program directive. It is similar to vsect, except it does not allocate any space. A csect, or constant section directive, is merely a convenient way of assigning successive values to a list of labels and is similar to the EQU directive. In Listing 2. fn.name has the value zero and fn.ext equals eight. You can also have more than one csect in a psect. If you specify an expression after the csect. this value becomes the base address for the section. In the second csect. first equals 10, second equals 11 and last equals 12.

Don't put your variables in a csect. even though one of the sample programs in the RMA manual does. Always use a vsect so the linker can properly assign values to the symbolic names. Remember that csect is used to define constants only. just like the EQU and SET directives. For example,

```
False   EQU 0
True    EQU 1
```

is the same as

```
csect
False   RMB 1
True    RMB 1
ends
```

With this in mind, the constants defined within a csect should normally be used with immediate addressing. For example, you can use the following source code with constants:

```
LDA     #True
CMPA    #True
```

The Four Golden Rules

When writing assembly-language programs for OS-9, keep in mind that you are writing for a multiuser, multitasking environment. This means your program does not have the degree of control you are used to with EDTASM+. The operating system determines where in memory your program runs and where its data storage area is. Also, the operating system must be free to interrupt your program and pause it so that the other active tasks receive their time, too. Unless your program has timing-critical code, leave the interrupts unmasked.

For all of this to work smoothly you must follow a few simple rules. First, you must write position-independent code. Your program must be able to execute from any position in the microprocessor's 64K address space. The only way to ensure this is to avoid using absolute addressing. This means that when you need to access strings, variables or constants in your program, you must specify them as an offset from the program counter (PC) register if they are in a psect or as an offset from Register U if they are in vsect and not as an absolute address. That is. instead of using either

```
ldy #name
jmp label
```

use

```
leay name,pcr
bra label
```

This is called position independent code and is a requirement in OS-9. The assembler takes care of the calculations for you. so the process is painless and soon becomes second nature.

Second, your program should never modify any memory outside its variable space. Since OS-9 allocates memory for each process at run time, your program might overwrite itself, or some other important data, if it is allowed contact with memory that does not belong to it. Remember, your programs are no longer running alone. Multiple copies of your program can be running at the same time. Therefore, your program must never modify itself.

Third, when your program begins, OS-9 gives it some important data located in the 6809's registers. With this data, a program knows where its variable storage is and whether or not any parameters have been passed to it. Some programs don't need to know this information, but you should make sure you save it, or use it. before loading the registers with other data.

Fourth, never directly access the CoCo's hardware. You should always use a system call to handle the desired function. This ensures your program runs on OS-9 systems other than your own. OS-9 has built-in routines for keyboard, screen, printer, and disk I/O, so it is a waste of time to write your own.

Interfacing with the Shell

Now. the $20.000 question: If you don't know in advance where your variable storage is, how do you access it? Fortunately. OS-9 provides enough data, via the registers, to provide a simple solution.

When OS-9 forks a process (which is what your completed program is when it executes), it puts the following data into the registers:

B — contains the number of characters in the parameter list, including the carriage return at the end.

X — points to the first byte of the parameter list.

Y — points to the top of the process' memory area, which is the end of the parameter list.

U — points to the bottom of the memory area, which is the first byte of the direct page.

DP — contains the most significant byte of the direct-page address. Never change the direct-page register.

Terminate your programs by using the F$Exit call. The value in Register B at the time of the F$Exit call is returned to the Shell as an error number. A value of zero indicates no errors. Make sure to load Register B with the proper value before calling F$Exit.

Since the direct-page register is set for you, any variables in the direct page can be accessed with direct addressing or as offsets from Register U. as long as it is not changed. Using the variables from Listing 2, you could use

```
lda path
stx count
```

Variables that are off the direct page should always be referenced as offsets from Register U to ensure your programs are using the memory space OS-9 has assigned to them. To ensure this process, use

```
leax IObuff,u
leay filename,u
```

instead of

```
ldx #IObuff
Idy #filename
```

Following the proper rules becomes automatic after a short while. Pay particular attention to the sample programs, since they demonstrate important principles.

Building An Executable Program

Program development with the Development System is a two-step process. First, the assembler gathers your source code into a Relocatable Object File (ROF). Then, the linker processes one or more ROFs. adds the proper module header and CRC bytes, and creates the final product. Although the two-step process for generating an executable module may seem tedious, it is this very process that makes RMA so powerful.

The power of this method comes from the fact that any symbolic name not defined in a psect is passed to the linker, which attempts to resolve it based on information from other psects. Thus, program code in one psect can use symbolic names defined in another psect.

To make this process easier to implement, the assembler gives you the ability to declare your symbolic names as either local or global names. Local names are defined only in their psect and are unknown to other psects. In fact, other psects may use the same names for different values without any "multiply defined symbol" conflicts.

Global symbols are made available to all psects. but they are referred to only if the psect has not also defined the same name. If one psect defines the symbol buffer as a global symbol and a second defines buffer as a local symbol, the symbol in the second psect is local and different from the global symbol of the same name.

To make a symbol global, include a colon (:) after the symbol name when it is defined. All other symbols are local symbols by default.

When using RMA, it is important to note that the assembler is case-sensitive. Thus, the labels Buffer, buffer and BUFFER are all different, even though they are spelled the same. Avoid the temptation of using the same name with different upper- and lowercase combinations in the same psect. Doing so is asking to become easily confused when debugging time comes around.

When multiple psects are linked, there must be one. and only one, mainline code segment. The mainline code segment is that section of code OS-9 transfers control to when the program begins executing. The psects in listings 1 and 2 are mainline psects. psects used with the mainline psect should be written as subroutines. They can have their own vsects, but they should not attempt to define the typelang. attrev. edition or entry bytes of the module. Make these entries in the psect statement zero or the linker refuses to link the code sections together.

Using Multiple psects

Direct your attention to the source code in listings 3, 4 and 5. These three psects, when linked together, form a program that waits for a keypress from the keyboard. dumps the values of the 6809 registers to the screen, and prints the ASCII value, in decimal, of the key pressed. Each listing is a separate file and is assembled separately from the others.

Notice that each psect has a symbol named buffer that labels a non direct-page

Offset	Size	Description
0,s	2 bytes	return address — don't change it
2,s	2 bytes	number of parameters passed
4,s	2 bytes	pointer to first parameter
6,s	2 bytes	size, in bytes, of first parameter
8,s	2 bytes	pointer to second parameter
10,s	2 bytes	size, in bytes, of second parameter

Figure 1: A Sample Data Block

Type	Size	Comments
BOOLEAN	1 byte	$FF= true. 0= false
BYTE	1 byte	
INTEGER	2 bytes	if left-most bit set, number is negative
REAL	5 bytes	8 bit exponent, 31 bit mantissa. 1 sign bit
STRING	varies	terminated with $FF if less than max size

Figure 2: The Five Basic Data Types

variable. These three symbols are all local to their psects and, thus, have different values.

There are three global symbols in this group of psects: Dec. RegDump and Divide. There is also only one mainline psect — main. Notice that the other psect directives specify the typelang, attrev, edition and entry bytes as zero.

Variable names that have been declared as global symbols can be accessed from other psects simply by using that symbol name in your source code. Keep in mind that the assembler passes only unresolved references to the linker, so if the code in a psect defines a symbol with the same name as a global symbol, the global symbol is not used.

To access a global variable from another psect. use one of the following methods. If the variable is a non direct-page variable, access them in the normal manner:

```
leay IObuf,u
ldx counter,u
```

If a global direct-page variable is to be accessed, precede the variable name with the less-than symbol (<):

```
lda <Accum
stx <pointer
```

RMA is smart enough to know when to use direct addressing for direct-page variables in its own psect. but symbols defined elsewhere are assumed to be extended (16 bit) addresses. Using the < forces the assembler to generate object code that uses direct addressing. Again, you can access variables as offsets from Register U, which also works for the direct-page variables.

Armed with this information, you can build a library of common routines, debug them once, and then use them repeatedly. Each library routine can declare its own variable space, so you can "set it and forget it" since the linker automatically allocates enough space when you link files together. This gives you the ability to write "black box" subroutines that perform their function without affecting the other parts of your program.

A Sample Assembler Session

Using the source code files in listings 3, 4 and 5 you would make an executable file by following these steps:

— Use a text editor to create the files demo.a, AsciiConv.a,and RegisterDump.a.

— Generate three ROFs. demo.r, AsciiConv.r, and RegisterDump.r in the following manner:

```
rma -o=demo.r demo.a
rma -o=AsciiConv.r AsciiConv.a
rma -o=RegisterDump.r RegisterDump.a
```

— If the file sys.l is in the current data directory, you can use the following command to invoke the linker and generate an executable file:

```
rlink -l=sys.l  -o=demo demo.r AsciiConv.r RegisterDump.r
```

—If the sys.l file is not in the current data directory, specify the complete pathlist after the l option as follows:

```
rlink -l = /dd/lib/sys.l  -0 = demo demo.r AsciiConv.r RegisterDump.r
```


Now, you should have a file called demo in your current execution directory. Notice how the file with the mainline psect is specified first in the ROF list for the linker. Other than that, it doesn't matter in what order you specify the modules, as long as the mainline section is first.

A quick word on the sys.l file. This file is a library file and is an ROF generated from the file defs/os9defs.a. You can insert the use /dd/defs/os9defs.a directive in each of your source code files and have the assembler resolve the OS-9 symbol constants, which slows the assembly process, or you can have the linker resolve all of the symbols in one step during the linking process. The second method is faster; the choice is yours.

OS-9 Level II

Listing 1: first.a

```
* listing 1

        psect first,$11,$81,1,100,Start

Start   leax   name,pcr       get prompt
        ldy    #name.len      and its length
        lda    #1             get standard output path number
        clrb
```

```
        os9    I$WritLn        output prompt
        os9    F$Exit          return to shell

name     fcc    /Have a nice day./
        fcb    13
name.len equ    *-name
```

Listing 2: second.a

```
* listing 2
* sample vsects and csects - not executable

Type     equ    $10
Lang     equ    $01
Att      equ    $80
Rev      equ    $01
Edtn     equ    $1
Stack    equ    2

        psect second,Type+Lang,Att+Rev,Edtn,Stack,Entry

        vsect  dp
count    rmb    2
path     rmb    1
        endsect

        vsect
IOBuffer rmb    256
filename rmb    11
        endsect

        csect
fn.name  rmb    8
fn.ext   rmb    3
        endsect

        csect 10
first    rmb    1
second   rmb    1
last     rmb    1
        endsect

Entry    code
        etc.
        endsect
```

Listing 3: demo.a

```
* listing 3

        psect main,$11,$81,1,100,Entry

        vsect
buffer   rmb    2
        endsect

Entry    lbsr   RegDump         dump registers to screen
        leax   prompt,pcr
        ldy    #prompt.l
        lda    #1
        os9    I$Write
        leax   buffer,u        get input buffer address
        ldy    #1
        lda    #1
```

```
        os9    I$Read          read from std in
        ldb    ,x              get character read
        clra
        lbsr   Dec             convert character to decimal ascii
        lda    #'=             insert '=' at start of string
        sta    ,-x
        leay   1,y             add one to string length
        lda    #1
        os9    I$WritLn        write ascii string to screen
        clrb                   tell shell no errors
        os9    F$Exit
prompt   fcc    /Press a key: /
prompt.l equ    *-prompt
        endsect
```

Listing 4: AsciiConv.a

```
* listing 4

* Function
*   subroutine Dec converts a 16-bit, unsigned binary number to a 5-digit
*   ASCII string terminated with a linefeed character
*
* Usage
*   call from the mainline code by
*
*       bsr Dec
*
* Entry
*   D holds the 16-bit number
*   U points to dp base address
*
* Exit
*   X points to the first byte of the ASCII string
*   Y contains the ASCII string length

        psect   AsciiConv,0,0,0,32,0

        vsect   dp
Accum   rmb     2
        endsect

        vsect
buffer  rmb     6
        endsect

Dec:    pshs    u
        leax    buffer+5,u      move to assumed end of buffer
        ldy     #1
        leau    Accum,u         get address of accumulator
        std     ,u              save number in accumulator
        lda     #13
        sta     ,x              mark buffer end
DecLoop ldb     #$10            load divisor
        bsr     Divide          do division
        adda    #'0             convert remainder to ASCII
        sta     ,-x             save in buffer
        leay    1,y
        ldd     ,u              all done?
        bne     DecLoop         branch if not
        puls    u,pc
```

```
* following subroutine divides an unsigned 16-bit number by a positive 8 bit
* entry: B = divisor
*       U points to 16-bit number
*  exit: A = remainder
*       U points to result

Divide: pshs    b,x
        ldb     ,u              get first byte
        clra                    clear remainder
        bsr     DoDiv           divide
        stb     ,u              store result
        ldb     1,u             get second byte
        bsr     DoDiv           divide
        stb     1,u             store result
        puls    b,x,pc
DoDiv   ldx     #8              loop counter - 8
DivLoop aslb                    rotate D left
        rola
        cmpa    2,s             is A larger than divisor?
        blo     TooBig          branch if not
        incb
        suba    2,s
TooBig  leax    -1,x            done all 8 repetitions?
        bne     DivLoop         branch if not
        rts
        endsect
```

Listing 5: RegisterDump.a

```
* listing 5
*
* OS-9 Register Dump
* Written 89/07/06 by Jeff Mikel
*
* Call using a BSR to RegDump
*
*  Entry: U points to dp base address
*
*  Exit:  register dump displayed on terminal
*         all registers are preserved

        csect
regs.cc rmb     1
regs.a  rmb     1
regs.b  rmb     1
regs.dp rmb     1
regs.x  rmb     2
regs.y  rmb     2
regs.u  rmb     2

        psect   RegisterDump,0,0,0,64,0

        vsect
buffer  rmb     80
        endsect

RegDump: pshs   cc,a,b.dp,x,y,u save registers on stack
        leax    buffer,u        get output buffer address
        leay    data,pcr        get prompt address
        bsr     copy            copy first reg name to buffer
        lda     regs.cc,s       get CC register
        bsr     hexout          convert
        lda     regs.a.s        get A
        bsr     hexout          convert
```

```
        lda     regs.b,s        get B
        bsr     hexout          convert
        lda     regs.dp,s       get DP
        bsr     hexout          convert
        ldd     regs.x,s        get X
        bsr     double          convert
        ldd     regs.y,s        get Y
        bsr     double          convert
        ldd     regs.u,s        get U
        bsr     double          convert
        tfr     s,d             get S
        addd    #return+2       compute S before subroutine call
        bsr     double          convert
        ldd     return,s        get return address
        bsr     double          convert
exit    lda     #$0d            flag end of buffer
        sta     ,x
        leax    buffer,u        get buffer start
        ldy     #80             get max output characters
        lda     #1
        os9     I$WritLn        display on terminal
        puls    cc,a,b,dp,x,y,u,pc restore registers and return

doulble bsr     HexConv         convert first byte
        tfr     b,a             get LSB
hexout  bsr     HexConv         convert second byte
        lda     #$20            insert a space in buffer
        sta     ,x+
copy    lda     ,y+             copy next prompt
        beq     retn
        sta     ,x+
        bra     copy
retn    rts

HexConv pshs    a
        rept    4               convert MS nibble to ascii
        lsra
        endr
        bsr     Digit
        lda     ,s              get byte again
        anda    #$0f            convert LS nibble to ascii
        bsr     Digit
        puls    a,pc

Digit   cmpa    #9              in range 0-9?
        bls     ToAscii
        adda    #7              digit is A-F
ToAscii adda    #$30            make it ascii
        sta     ,x+             save in buffer
        rts

data    fcc     /cc=/
        fcb     0
        fcc     /a=/
        fcb     0
        fcc     /b=/
        fcb     0
        fcc     /dp=/
        fcb     0
        fcc     /x=/
        fcb     0
        fcc     /y=/
        fcb     0
        fcc     /u=/
        fcb     0
        fcc     /s=/
```

```
        fcb     0
        fcc     /pc=/
        fcb     0
        fcb     0
        endsect
```

Listing 6: AddInteger.a

```
* listing 6

        psect   AddInteger,$21,$81,1,100,Start

        csect
return  rmb     2               Basic09 return address
pcount  rmb     2               number of parameters passed
param1  rmb     2               pointer to first parameter
psize1  rmb     2               size of first parameter
param2  rmb     2               pointer to second parameter
psize2  rmb     2               size of second parameter
param3  rmb     2               pointer to third parameter
psize3  rmb     2               size of third parameter
        endsect

Start   ldb     #100            prepare for possible error
        ldx     #3              exactly three parameters passed?
        cmpx    pcount,s
        bne     error           branch if not
        ldx     #2              each parameter size of type integer?
        cmpx    psize1,s        check first parameter
        bne     error           branch if wrong size
        cmpx    psize2,s        check second parameter
        bne     error           branch if wrong size
        cmpx    psize3,s        check third parameter
        bne     error           branch if wrong size
        ldd     [param1,s]      get value of first parameter
        addd    [param2,s]      add to value of second parameter
        std     [param3,s]      store as value of third parameter
        clrb                    signal no errors
        rts
error   orcc    #1              set error flag
        rts
        endsect
```

Listing 7: ParamDump.a

```
* listing 7

        psect   ParamDump,$21,$81,1,100,Start

        csect
        buffer  rmb     3       I/O buffer
        return  rmb     2       Basic09 return address
        pcount  rmb     2       number of parameters passed
        param1  rmb     2       pointer to first parameter
        psize1  rmb     2       size of first parameter

Start   leas    -return,s       set up variable space on stack
        ldb     #100            prepare for posiible error
        ldx     pcount,s        get parameter count
        cmpx    #1              exactly one?
        bne     error           exit with error if not
        leax    buffer,s        get i/o buffer address
        ldb     #32             put space character at end
        stb     2,x
```

```
        ldy     psize1,s        get parameter size
        ldu     param1,s        get parameter location
loop    pshs    y
        lda     ,u+             get next byte of parameter
        pshs    x
        bsr     ConvHex         convert to Ascii hex
        puls    x
        lda     #1              standard output path
        ldy     #3              number characters to write
        os9     I$Write
        puls    y
        leay    -1,y            done entire parameter?
        bne     loop            repeat if not
        ldb     #13             else send carriage return to output path
        stb     ,x
        os9     I$WritLn
exit    leas    return,s        restore stack
        orcc    #1              indicate error occurred
        rts

ConvHex pshs    a
        rept    4               get four high bits
        lsra
        endr
        bsr     Digit           convert to Ascii digit
        puls    a
        anda    #$0F            get four low bits
Digit   adda    #'0             make Ascii number digit
        cmpa    #'9             outside range 0-9?
        bls     legal           branch if not
        adda    #7              else make in range A-F
legal   sta     ,x+             save in i/o buffer
        rts
        endsect
```

Listing8: Tester.b09

```
PROCEDURE  Tester
0000    (*  listing 8
000C    (*  illustrates  how to call   the machine-1anguage  subroutine
0046    (* IntAdd that is made from listing six
006D    DIM  first,second,sum:INTEGER
007C    first=1991
0084    second=2001
008C    RUN IntAdd(first,second,sum)
00A0    PRINT   first;   "  +  ";   second:   " - ";   sum
```

Listing 9: ParamTest.b09

```
PROCEDURE ParamTest
0000    DIM  f:REAL
0007    DIM  i:INTEGER
000E    DIM  b:BYTE
0015    DIM  bo:BOOLEAN
001C    DIM  S:STRING[20]
0028
0029    f=123.45
0034    i=12345
003C    b=123
0043    bo=TRUE
0049    s="This is a string"
0060
0061    PRINT "REAL     =  ";
0070    RUN ParamDump(f)
```

```
007A    PRINT "INTEGER  =  ";
0089    RUN ParamDump(i)
0093    PRINT "BYTE     =  ";
00A2    RUN ParamDump(b)
00AC    PRINT "BOOLEAN  =  ";
00BB    RUN ParamDump(bo)
00C5    PRINT "STRING   =  ";
00D4    RUN ParamDump(s)
00DE    END
```

To execute the program, type demo at the OS9: prompt. To see how the hardware registers change, try adding some parameters after demo on the command line. For example, try demo abcd, demo #16K and demo abcd #16K.

Interfacing with BASIC09

The rules for writing machine-language programs for BASIC09 are somewhat different. These programs execute as subroutines, not as individual processes, so your program development is different in several ways. First, the typelang byte in the psect directive must be set to $21 instead of $11. Second. BASIC09 does not allocate memory forth subroutine, so your subroutine has severe restrictions on memory usage for its variables. All communications between your subroutine and your BASIC09 program are done via the hardware stack (the S register). Finally, and of great importance, your program must end with an rts instruction, not the F$Exit call. Using F$Exit terminates BASIC09 as well. Your programs must still be written in position-independent code. Programs not written in position-independent code do not work with BASIC09.

When BASIC09 calls your machine-language subroutine. Register S points to a data block your subroutine can access. A sample data block might appear as shown in Figure 1.

The size of the data block varies depending on the number of parameters sent. If no parameters are sent, the data block contains just the return address and the parameter count, which would be zero in this case.

Each parameter is defined by a pointer to the area used to store the parameter and a count of how many bytes are required for storage. Therefore, each takes up four bytes on the stack. Any BASIC09 data type can be sent as a parameter to your subroutine. The five basic data types, and their sizes, are shown in Figure 2.

The size of a string variable will always be the number of bytes specified in the DIM statement, not the actual length of the string stored in that section of memory. If the length of the string is shorter than its dimensioned size, which is 32 bytes unless specified otherwise. the string is terminated with $FF(255 decimal). You must find the length manually by searching for this character. Also, make a note not to store the parameter values directly on the stack rather they are accessed indirectly. See the sample programs for more clarification.

At the completion of your subroutine, you may return an error code to BASIC09 by loading Register B with the desired error number and setting the carry flag. If the carry flag is not set. BASIC09 assumes no errors occurred, regardless of the contents of Register B.

Your subroutine knows only the size and location of a parameter passed to it. There is no type checking, so your subroutine is not able to discriminate between variables of different types that are the same size. For example, assume the following statements appear in a BASIC09 program:

```
DIM name:STRING[5]
DIM mass:REAL
DIM state(5):BOOLEAN
```

Also, assume you have created a machine-language subroutine called Sample. which can be executed from a BASIC09 program through the following command:

```
run Sample(parameter)
```

You could execute Sample in any of these three ways:

```
run Sample(name)
run Sample(mass)
run Sample(state)
```

In each of the above cases, the subroutine Sample will not know which of the three BASIC09 variables was sent. Naturally, Sample expects a certain variable type — STRING, REAL or BOOLEAN — but it does not know whether the parameter sent to it is of the proper type. You needn't worry too much about this situation since the contents of a variable of the wrong type will, in all likelihood, look like garbage. Just keep in mind there is no way to explicitly check a variable's type.

As a matter of fact, the same is true for BASIC09 procedures. If a calling procedure sends as its argument a REAL (floating-point) number, the procedure called can declare its parameter as a REAL, a five-element array of type BYTE, or a five-element array of type BOOLEAN (try it!). No errors are generated because there is no type checking. Any problems that arise are as a result of the way the called procedure interprets the data sent to it. Again, this is not something you have to worry about. You just need to know you cannot check a parameter's type. Rather, just check for the expected size and for valid data and all will be fine.

Program Memory Under

As stated earlier, your subroutine will not have any memory reserved for it. Very few programs can keep track of all the relevant data in the CPU registers, so you must have some memory somewhere. Where do you get it? There are two ways to solve this problem.

First, your program can reserve some extra space on the hardware stack. For example, if your program needs 10 bytes of variable storage space, you can use the instruction

```
leas   -10,s
```

at the beginning of the program. Then, at the end, you'll need to "bump" the stack pointer back to its original position. Exit the subroutine using

```
leas  10,s
rts
```

This method is simple, but it does not allow your subroutine to have any static variables, which means all variables are erased when your subroutine ends. They do not exist except when your subroutine is running and they cannot be saved for later use by the subroutine.

If you need static variables, meaning that they don't "go away", you can use a BASIC09 variable as storage. For example, the 10 bytes of storage space used above could also be stored in an array declared by the BASIC09 program:

```
DIM memory(10):BYTE
```

Then, so that the machine-language subroutine has access to the space occupied by memory, use

```
RUN Sample(memory)
```

Subroutine Sample can then use the 10 bytes of Variable memory for its variables. Keep in mind that variables in BASIC09 are local to the procedure that defines them. Therefore, unless explicitly passed from one procedure to the next, any static storage space is also local. Also, the static storage space exists only while the BASIC09 procedure that calls your machine-language subroutine is running, which makes it dynamic storage just like all other BASIC09 variables. This is also why the word static is italicized. In the true sense of the word, BASIC09 does not offer any static variables, so be careful when using this technique.

It is important to note that when writing BASIC09 subroutines, your program is not the primary module, thus it should not attempt to adjust the amount of memory set aside for the process. BASIC09 is in charge. You must work with it. not around it.

Listing 6 is a simple subroutine that takes three parameters, all integers. The first two integers are added together and returned via the third parameter. This program is fairly straightforward and is intended to illustrate how to access and change the parameters. Notice the use of a csect to keep track of all of the offsets for accessing the parameters. This way. the offsets are automatically computed, which saves you time, trouble and prevents mistakes due to incorrect offsets.

Executable programs for BASIC09 are created in the same manner: assemble and link them. BASIC09 automatically loads them when they are called, provided the subroutines are in the current CMDS directory. For example, the instructions to make an executable module out of Listing 6 are as follows:

— Create the file Addlnteger.a with a text editor.

— Assemble AddInteger.a to a ROF, Addlnteger.r using:
rma -o=Addlnteqer.r Addlnteqer.a

— Use the linker to generate an executable module using:
rlink -o=IntAdd Addlnteger.r

— From BASIC09. call the subroutine in the same manner as you would any BASIC09 subroutine: run IntAdd(x1, x2, x3)

where x1, x2 and x3 are all integers. Run Listing 8 for a real-life sample of this routine in operation.

Listing 7 is another simple subroutine. It gives a hexadecimal dump of the contents of a single parameter. Use this subroutine to examine how different data types look in the computer's memory. This program uses the stack for all of its variable storage. Notice again the use of a csect to keep all of the offsets straight. Also notice the stack pointer is restored to its original value before that final rts instruction. This is a must, but it is easy to forget. Assemble and link ParamDump.a (Listing 7) in the same manner as you did for Listing 6 and then run the program in Listing 9 to see the program in action.

Be careful when writing programs for BASIC09. Remember, do not attempt to write to memory that has not been given to you by a BASIC09 procedure. Be especially careful if you are rewriting a program for BASIC09 that ran under the Shell. Remember you do not have access to the direct page — that belongs to BASIC09. Careful planning prevents any mishaps.

Hopefully, you will find that assembly-language programming under OS-9 is enjoyable, especially since you do not have to worry about writing disk and keyboard drivers or Hi-Res graphics routines anymore. Plus, if you follow the rules, your programs automatically work with all other OS-9 programs. This is all made possible by OS-9's standard file structure and graphics routines. I believe once you start programming in assembly language under OS-9, you won't want to go back to Disk BASIC.

I would like to think this article has answered all your OS 9 assembly-language questions, but I know it hasn't. Please feel free to send me any questions or comments.

Jeff Mikel is a recent graduate of Georgia tech. where he studied electrical and computer engineering. He has been CoCo programming for nine years. Jeffs other interests include hi-fi audio, electronics and outdoor photography. He can he contacted at 3300 Gate Court. Rex. GA 30273. Please include an SASE when requesting a reply.